A Distributed Component Framework for Science Data Product Interoperability

Daniel Crichton, Steven Hughes, Sean Kelly, Sean Hardman
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, California 91109

Abstract - Correlation of science results from multi-disciplinary communities is a difficult task. Traditionally, data from science missions is archived in proprietary data systems that are not interoperable. The Object Oriented Data Technology (OODT) task at the Jet Propulsion Laboratory is working on building a distributed product server as part of a distributed component framework to allow heterogeneous data systems to communicate and share scientific results. These components communicate using a standard metadata interchange language. This provides an excellent vehicle for turning data into information and allowing for data in unique formats to be correlated and exchanged. Advances in Internet and distributed object technologies provide an excellent framework for sharing data across multiple data systems. The product server component of the OODT framework allows for results to be interchanged between native data system formats and the framework using an XML-based query language. The product server component wraps data system interfaces, which abstracts away the data system unique interfaces, and provides a scalable architecture by providing query handlers that facilitate the interchange of queries and results. This paper, the second in a series on the OODT task [4], focuses on the development of the product server component using the Planetary Data System (PDS) as an example system. This continues the discussion of an enterprise framework that allows for data system interoperability across multiple science disciplines.

I. Introduction

Science data has continued to devolve into a large set of highly fragmented distributed data systems. These systems are heterogeneous and geographically distributed making interoperability and integration difficult. Furthermore, correlating science data across a multi-disciplinary environment is even more challenging. The Object Oriented Data Technology task at the Jet Propulsion Laboratory is currently researching a distributed framework that will allow for data products from distributed data systems to be located and retrieved.

Data systems across NASA are heterogeneous in nature. They have traditionally followed stovepipe implementations, and there has been very little integration across these systems. The implementations are often unique, and there is no standard mechanism for data interchange between these systems. A key to linking these data systems together, however, is metadata. The NASA Planetary Data System (PDS) has developed an archive standards architecture that provides a good metadata foundation that can be used to build an interoperable data architecture for exchanging data products for planetary missions.

The Planetary Data System (PDS) manages and archives planetary science data for NASA's Office of Space Science. In existence since the late 1980s, the PDS early on developed a standards architecture that included a formal enterprise model, a means for collecting and associating metadata with science data products, and a peer review process for ensuring data and metadata validity. This active science data archive currently has over five terabytes of data curated by six geographically distributed science nodes and stored and distributed on CD and DVD media. The standards architecture has proven to be critical to maintaining consistency across the various science domains represented in the planetary science community and for supporting a level of interoperability across the nodes. The products are stored in a variety of data and machine formats and are served from heterogeneous hardware platforms. For example, the archive includes a diversity of data products from images and time series to spectrum. However the metadata used for searching and describing the resulting data products is consistent due to the standards architecture.

A key to linking enterprise data systems and databases is to provide a common metadata model. The PDS experience in developing and enforcing metadata standards has proven to be a critical element in providing a relationship across the distributed science nodes. The advent of the Internet and web technology now affords an excellent opportunity to build an interoperable framework that will allow systems to exchange data based on metadata. The OODT task is currently working with the Planetary Data System to provide a data architecture that allows for data products within the PDS to be located and exchanged across the distributed nodes using a common user interface. We also feel that the methodology being developed will directly address the data system interoperability problems now being encountered in general.

II. Architecture

The OODT architecture has several key architectural objectives which include (1) requiring that individual data systems be encapsulated to hide uniqueness; (2) requiring that communication between distributed services use metadata for data interchange; (3) defining a standard data dictionary based on metadata for describing data resources; (4) providing a solution that is both scalable and extensible; (5) providing a standard mechanism for exchanging data system product results across distributed services; and (6) allowing systems using different data dictionaries to be integrated.

The data architecture that is being built by the Object Oriented Data Technology task is focusing on building a middleware[1] component framework that provides the infrastructure necessary to interconnect and encapsulate data systems. The product service, the focus on this paper, is part of a larger component framework [4], which includes a query, profile and product server. Profile and product service instances are distributed across the enterprise and manage information and access to a set of data system resources. A key benefit of this architecture is that new service instances (or servers) can be introduced in order to scale the system. The basic system architecture is illustrated in Figure 1.

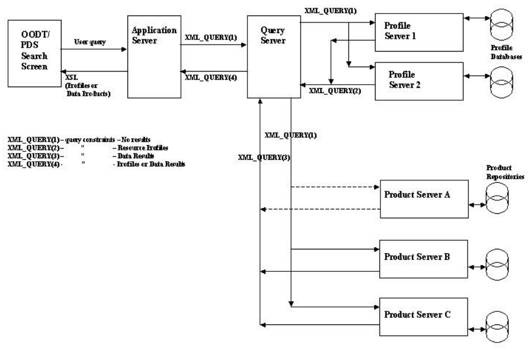


Figure 1: Example System Architecture

The profile server manages profiles—sets of resource definitions [11]—about distributed data systems and their products. A profile is a metadata description of the resources known by a node in the distributed framework. These resources are interfaces, data products, or profiles available in the integrated enterprise. Profiles may be grouped and served by more than one profile server. The query component ties this architecture together by providing and managing the traversal of the integrated digraph node architecture. It also interprets profile definitions that provide mappings between data system nomenclature. The query component also provides the facility to manage concurrent queries across multiple servers in order to improve performance.

The product server provides the translation necessary to map a product retrieved from a data-system–dependent environment into a neutral

[1] In the computer industry, middleware is a general term for any programming that serves to "glue together" or mediate between two separate and usually already existing programs [http://www.whatis.com].

format suitable for exchange between systems. The product server architecture is similar to the profile architecture by providing a distributed approach allowing one or more instantiations of product servers across a distributed enterprise. A specific goal of this architecture is to allow heterogeneous data systems to be easily added without changing the way their data is stored.

The distributed architecture described lends itself naturally to a distributed object implementation. We used the Common Object Request Broker Architecture (CORBA) to provide the distributed object framework and to communicate and exchange data in heterogeneous environments using the Internet Inter-ORB Protocol (IIOP) [17]. This activity is currently using an implementation of the CORBA 2.0 standard from Object Oriented Concepts known as Orbacus [18]. Each profile and product server node is defined by a separate object name (or node name). The CORBA naming service allows for nodes to be located using the naming service that is included in the Orbacus implementation. Profile and product server instantiations are uniquely identified by name. These names are used as part of the metadata header encoded to identify enterprise services that can support queries for distributed products.

Components of the architecture communicate using an Electronic Data Interchange (EDI) mechanism implemented using the Extensible Markup Language (XML) [21]. The use of XML, as part of the data interchange, isolates the data content and data transport. This is significant since it allows both the interchange language and the transport mechanism to evolve independently. It also continues to allow the architecture to be focused on metadata development.

The XML interchange language that is implemented is referred to as the OODT Query Definition. The query definition is implemented independent of any one database, functional, or programming language and is intended to provide an abstract view of both the query expression and the results. Using a query definition as the interface between the services allows for each component (profile or product server) to plug into the architecture by either satisfying queries that it receives, or returning a null set. The query definition also allows for each data system to be encapsulated. This allows various implementations ranging from the use of relational and objects database management systems to implementations that use flat file and home-grown databases for cataloging and storing data products to exchange information by plugging into a generic query definition.

The component framework has been developed entirely in the Java programming language along with CORBA and XML [21]. Java allows for the implementation of the object architecture and allows the framework to be easily extended to integrate new data systems. Java is particularly useful in the design of the product service component due to its ability to dynamically bind objects at run time which allows the system to easily integrate new data systems.

One of the goals of this architecture is to provide a standard application program interface (API) that will allow for generic science analysis tools to be written that can plug into the architecture to retrieve and correlate data from multiple data sources. This is accomplished through the component architecture. Abstracting the implementation away from the client allows for the infrastructure to evolve and expand without breaking the tool interfaces. It also moves the domain intelligence to the middleware components which removes the constraint that the tools need to have the knowledge of the protocol and location of data systems in order to query and retrieve data from them. The use of a component framework also allows for new server-side components to be developed in order to extend the query capabilities.

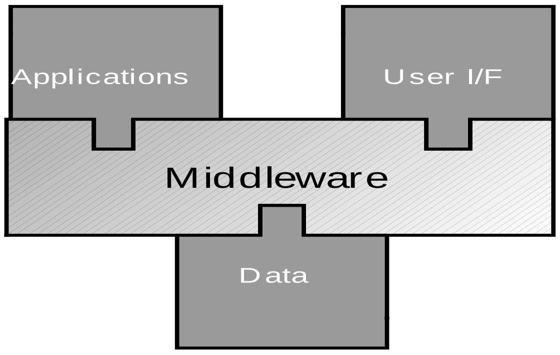


Figure 2: Role of Middleware

In Figure 2 the framework shows a basic middleware architecture with clients and servers plugging together.

The data architecture described focuses on providing a framework for solving complex integration problems across heterogeneous data systems. It addresses the issues of data location, data transformation, and data exchange. The framework provides a scalable architecture that centers on the use of metadata. It also allows for data systems to continue to retain their unique attributes, yet plug into an enterprise data architecture that allows for the successful exchange of data content through the use of XML. By using XML this framework is able to impose an inter-disciplinary communication mechanism for data to be shared and exchanged.

III. Profile Service

Instruments and experiments generate science data products that are archived in data systems. Unfortunately, these data systems are heterogeneous and there are few common standards for querying. This makes locating data across these systems very difficult. Scientists and researchers are typically required to visit each data system independently and use local tools in order to locate the data. The proposed distributed component framework can be used to encapsulate each data system and thereby create a network of product servers. The profile service provides a common standard for locating the server that will resolve an incoming query.

The profile service uses resource profiles or metadata[2] to describe each product server and the data products it serves. Using these resource profiles, the profile server uses an incoming query's constraints to determine what product servers in the digraph of distributed product servers can resolve the query. For example, within a space sciences implementation of this framework, a query for selected images of Mars should be sent to the product server maintaining the Mars Global Surveyor images.

The Extensible Markup Language (XML) was chosen as the language for the resource profiles. The Document Type Definition (DTD) specification in Figure 3 illustrates the basic components of the resource profile which has three parts: the profile attributes, resource attributes, and the profile elements.

The <profAttributes> section provides attributes of the profile itself such as a system-wide unique identifier, title, and description. For a more complete description of the DTD elements, the DTD and a data dictionary have been registered with OASIS[16].

```
<!ELEMENT profiles (profile+)>

<!ELEMENT profile
(profAttributes,
 resAttributes,
 profElement*)>

  <!ELEMENT profAttributes
  (profId, profVersion*, profTitle*,
    profDesc*, profType*, profStatusId*,
    profSecurityType*, profParentId*,
    profChildId*, profRegAuthority*,
    profRevisionNote*, profDataDictId*)>

  <!ELEMENT resAttributes
  (Identifier, Title*, Format*,
   Description*,Creator*, Subject*,
   Publisher*, Contributor*, Date*,
   Type*, Source*, Language*,
   Relation*, Coverage*, Rights*,
   resContext*, resClass*, resLocation*)>

  <!ELEMENT profElement
  (elemId, elemName*, elemDesc*,
   elemType*, elemUnit*,
   (elemValue |
    (elemMinValue, elemMaxValue))*,
   elemSynonym*, elemObligation*,
    elemMaxOccurrence*,elemComment*)>
```

Figure 3: OODT Profile DTD

The <resAttributes> section provides the actual resource attributes. The Dublin Core metadata element set [9] for describing electronic resources has been adopted. Three additional resource attributes have been added to identify the resource's local discipline, location, and class.

The <profElement> section, the final part of the profile, provides a description of the resource's data content by providing a list of data product attributes and their values. For example, the Planetary Data System (PDS) maintains a DVD jukebox that provides online access to most of its archived data products. A resource profile for the jukebox as a product server would encode searchable attributes into this section of the profile. For example instrument types, target names, mission names and their associated

[2] Metadata is, literally, data about data, or information that describes the characteristics of data. For example, 37.6 is data. The fact that it's a measurement of a body's temperature in Kelvins is metadata. [6]

values would be included indicating that the server could handle a query with these attributes as constraints.

As can be seen from Figures 3 and 4, each data attribute is defined using meta-attributes such as <elemId> and <elemValue>. To maintain compliance with developing international standards, these meta-attributes are consistent with those defined in ISO/IEC 11179 – Specification and Standardization of Data Elements. The description of this standard is available through ISO/IEC.

```
<profile>
  <profAttributes>
    <profId>PROFILE_nnn</profId>
    <profTitle>Profile </profTitle>
    <profDesc>This profile … </profDesc>
  </profAttributes>
  <resAttributes>
    <Identifier>PDSPS_nnn</Identifier>
    <Title>PDS DVD Jukebox…</Title>
    <Format>image/pds</Format>
    <Language>en</Language>
    <resContext>NASA.PDS</resContext>
    <resClass>data.granule</resClass>
    <resLocation>http://</resLocation>
  </resAttributes>
    <profElement>
      <elemId>TARGET_NAME</elemId>
      <elemType>ENUMERATION</elemType>
      <elemValue>DEIMOS</elemValue>
      <elemValue>MARS</elemValue>
      <elemValue>PHOBOS</elemValue>
      <elemSynonym>ADS.OBJECT_ID
                     </elemSynonym>
    </profElement>
</profile>
```

Figure 4: Example Profile - PDS DIS

Under this approach, supporting interoperability between product servers from different domains is strongly dependent on metadata compatibility, or how well the metadata spans the domains. For example, two related domains such as planetary science and astrophysics both associate one or more target bodies with most data products. However, unless the same identifier is used to identify the target, even simple location of products cannot be easily supported. In fact as more sophisticated interoperability such as data transformation and correlation are requested, deeper levels of metadata compatibility will be required. For example, once a target body is identified, sufficient metadata must then be available for reference frame identification and possible conversion.

The profile service supports the location of products in two ways. Given a query, the profile first identifies resources that can resolve the query. The profiles of these resources can be returned directly to the application and the application can use elements of the profile to provide resource descriptions and links for the user. Alternatively, the framework can broadcast the query to the identified resources and return the results to the users. These two alternatives provide application developers with the flexibility to build robust product location services.

When using metadata to enable interoperability between domains, the hard problem of finding metadata commonalities across domains arises. This typically involves identifying similar attributes, determining core concepts, generalizing descriptions, and determining the primary name and synonyms. The profile service has started to address this problem through <elemSynonym>. This profile attribute simply maps synonyms to primary names.

IV. Query Architecture

The query service of the framework serves as the starting point for users to retrieve information stored across distributed data nodes. The query service's CORBA interface enables applications to have a programmatic entry point for entering queries and retrieving results. To facilitate application development, we have implemented a Java API that wraps the CORBA interface (a C++ API is forthcoming). This enables scientists and engineers to develop their own data analysis applications to access disparate data systems from a single API. In addition, as more data systems are added to the framework, existing applications can access the new systems with no changes. Furthermore, multiple user interfaces that access the query component are possible. One such interface that we have developed is a web interface. The web interface uses the Java API to give scientists and engineers immediate access to data systems from any common web browser without any programming or knowledge of what data systems to search.

Once a query as been entered, the query service first determines what resources registered within the system can resolve the query by searching a directed graph of resource profile nodes. The query service uses the CORBA naming service to connect to a profile node within the directed graph. In general, searches will enter at the root or parent node; however, the query service can enter and search starting at any node in the graph.

The query service "crawls" through multiple nodes in the directed graph automatically. The query service uses "spider" objects to execute queries at a node and are part of the scatter-gather approach: each object can run in its own thread of execution, maximizing the concurrency of multiple node searches in the system. The system scatters the spiders across nodes and gathers their results as they become available.

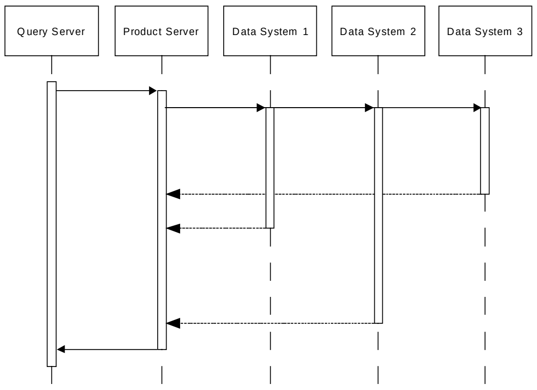


Figure 5: UML Diagram of Query Sequence

An Unified Modeling Language (UML) [2] sequence diagram of a typical search is illustrated in Figure 5. In the diagram, objects are shown across the top with their lifelines dropping down as time increases. Rectangles over the lifelines depict when an object is active. Solid arrows show method calls on an object, while dashed arrows show returns from those calls. A user’s query triggers the action at the Query Server object through its “execute search” method. The Query Server asks its root Profile Server object for any matches to the query. In response, the Profile Server returns three possible other Profile Servers that could contain matches (in addition to any matched resource profile it has itself). Concurrently, the Query Server executes the same query on the other Profile Servers. As each server returns more information, the Query Server may query yet more and more servers.

Since the directed graph of resource profiles is not necessarily acyclic, the query service must take care not to re-query profile nodes it has already visited, or else it could get caught in an infinite loop. The query component trivially prevents this by tracking a set of profiles it has queried so far.

Once the query component's spiders have completed their tasks and depending on query attributes the query service either simply returns the resource profiles or re-broadcasts the query to the identified resources so that they can in turn perform the query. In the former case, the query service collects the resulting resource profiles and returns them to the calling applications. This scenario is illustrated in Figure 1 when XML_QUERY(4) is returned to the application. The application can translate the resource profiles into HTML for presentation within a web browser. Using resource location information from the profiles, the application can create a hyperlink to allow a user to link directly to the resource.

If the query is re-broadcasted to the identified resources, each resource performs the query and returns the results to the query service. The query service then re-assembles the resulting collections into a single collection and returns it to the application. This scenario is illustrated in Figure 1 where XML_QUERY(4) returns product server results.

One possible extension that we are considering for the query service is to make it available via the HTTP standard. This would allow HTML pages to send XML queries directly to selected resources and render results directly into the HTML document.

Within the query service, a query is represented as an XML document. The document includes a header section that provides query attributes such as unique identifier, title, description, security level, and a revision note. Following these attributes are indicators as to how the query should propagate through the profile digraph, the maximum number of results allowed, number of results, and a string representing the original query. For example, the following XML document results from the input of the simple keyword query "TARGET_NAME = MARS".

```
<query>
 <queryAttributes>
```

```
  <queryId>OODT_Q070321</queryId>
  <queryTitle>PDS DIS Query</queryTitle>
  <queryDesc>PDS DIS Query Example</queryDesc>
  <queryType>QUERY</queryType>
  <queryStatusId>ACTIVE</queryStatusId>
  <querySecurityType>UNKNOWN</querySecurityType>
  <queryRevisionNote>2000-05-12</queryRevisionNote>
  <queryDataDictId>PDS_DS_DD_ V1</queryDataDictId>
 </queryAttributes>
 <queryResultModeId>ATTRIBUTE</queryResultModeId>
<queryPropogationType>BROADCAST
                                        </queryPropogationType>
 <queryPropogationLevels>N/A</queryPropogationLevels>
 <queryMaxResults>100</queryMaxResults>
 <queryResults>0</queryResults>
<queryKWQString>TARGET_NAME=MARS
                                        </queryKWQString>
 <querySelectSet></querySelectSet>
 <queryFromSet></queryFromSet>
 <queryWhereSet>
  <queryElement>
   <tokenRole>elemName</tokenRole>
   <tokenValue>TARGET_NAME</tokenValue>
  </queryElement>
  <queryElement>
   <tokenRole>LITERAL</tokenRole>
   <tokenValue>MARS</tokenValue>
  </queryElement>
  <queryElement>
   <tokenRole>RELOP</tokenRole>
   <tokenValue>EQ</tokenValue>
  </queryElement>
 </queryWhereSet>
 <queryResultSet></queryResultSet>
</query>
```

The actual query is encoded into three separate sections of the XML document. The <queryWhereSet> section encodes query constraint terms using post-fix notation. As can be seen in the example, the string "TARGET_NAME = MARS" has been encoded using the tokens "TARGET_NAME", "MARS", and "EQ" as <queryElements> together with tokens such as "LITERAL" to indicate roles. These tokens are used as constraints against a resource profile's profile elements.

The <querySelectSet> includes the names of elements to be returned from the query. Since no elements were specified in the example, the default is to return all elements.

The <queryFromSet> is encoded similar to the <queryWhereSet>, except that its elements are used as constraints against profile or resource attributes. For example, <resClass> could be used to return only profiles that describe system interfaces using the value "application.interface".

The <queryResultSet> collects the results produced by the query. As mentioned above, resource profiles are returned after the digraph search. After redirection of the query to selected resources, the results from the system resources performing the query are returned.

The Java class XMLQuery has been developed to manage the XML query documents. Currently there are two constructors for the class. The first accepts a simple keyword query such as "TARGET_NAME = MARS", parses the string and encodes the query in the structure. The second accepts a query in XML document format. Additional constructors can and will be implemented for other query formats. A method exists for returning the query in XML document format. Methods and iterators are also available for accessing any of the query elements and for putting and getting results.

The XMLQuery object is a key element of the distributed component framework. As a user query enters the system, it is encoded into an XMLQuery object by the query server and then sent to and used by the profile server component to search the digraph. All resulting profiles are collected in the object and returned to the query server. Depending on a query attribute, the collected profiles are either returned to the calling application or the original XMLQuery object is redirected to each identified resource. In the second case, the query results from each resource are collected in the object. The XMLQuery object is passed as a message within the system, however, the XML query document can be created for communicating with applications outside the system.

A key goal of the distributed component framework is enabling interoperability between heterogeneous data systems. As mentioned above, this framework uses system encapsulation, resource profiles, and message passing to hide system heterogeneity. This leaves the role of enabling interoperability to metadata.

In this context, metadata can enable interoperability in two ways. The first is to find and use metadata that is common to the separate data systems. For example, within the planetary data systems the data element TARGET_NAME is used to identify the object from which data is being collected. In astrophysics systems OBJECT_ID is used. If one of the terms were chosen and adopted throughout a system that encompassed both domains, a simple level of interoperability is trivially enabled. The second and more likely scenario is that both terms will be used interchangeably within the framework. The framework addresses this situation by allowing synonyms to be included in a resource's profile as illustrated in Figure 4. As can be seen, TARGET_NAME is included as an <elemName> and OBJECT_ID is included as its elemSynonym. This provides a mapping between the two terms that can be used by any service within the framework. For example, a planetary scientist could submit the "TARGET_NAME=MARS" query and the system would be able to locate and create a query for an astrophysics resource using OBJECT_ID=MARS.

The query architecture uses resource profiles to identify resources that can resolve a query. As described earlier, the resource described can be a product server that uses the query to retrieve and return data products. This implements a one-to-many mapping between resource profiles and data products. Alternatively, a profile can describe a single data product. This implements a one-to-one mapping between profiles and data products. Even though not efficient for large collections of data products, this latter approach provides a simple mechanism for implementing a data product inventory.

V. Product Server Architecture

The product service component, like the profile component, is instantiated as a node in the distributed architecture and provides the capability to return data system products based on a query. This allows each data system to maintain heterogeneous implementations, but still integrate into the enterprise architecture. Generally, administrators will start one product server for each data system, but such a server does not have to be collocated with its data system.

Product servers accept the same XML query structure as profile servers. Their response, however, is different. Instead of retrieving a profile or a portion of a profile that describes data system resources, it delivers actual products from a data system. Moreover, it converts the products from their underlying storage medium into a uniform transport format. These differences do not affect the structure of the XML document representing the query, which encapsulates the query, profile results, and product results. Product results appear under the queryResultSet tag.

Each product server consists of two parts: the enterprise interface and the data-system–specific interface, known as the query handler. The query mechanism uses the enterprise interface to make queries for products. Because this is a CORBA interface, like that of the profile servers, API developers can query a product server directly, although calling upon the query service is more common. The product server's enterprise interface then calls upon the data-system–specific query handlers to retrieve products from specific data systems. Developers integrating the OODT software into a new data system environment provide implementations of query handlers to communicate with specific, implementation-dependent data systems.

As mentioned, each product server node provides data access to one or more data systems. A product server node instantiates a Java-based server that integrates with the query service and receives XML-based queries using the XML query structure explained in Section IV. This profile attribute simply maps synonyms to primary names.

The product server framework that is provided is a generic Java-based server that dynamically loads query handlers defined and registered with the service. Once the server receives a query it then notifies each registered query handler as a separate thread managed by the product server. This allows the product server to time out queries to resources which may not be available as well as maximize concurrency to multiple data systems. The product server then packages the results from each query handler and returns the results using the XML-defined query definition. These results are then passed back to the Query Service which integrates all the results from the distributed product servers.

Query handlers provide a wrapper around each data system interface. This abstracts the data system away from the enterprise and allows the query handlers to function as a translation service. Developers implement query handlers using Java's type model, which separates types from classes using interfaces[3]. The query component specifies a standard Java interface to

[3] An interface in Java is a specification for the methods of a class. A class that implements a named interface must provide a definition for each method specified by interface or else be marked as an abstract class.

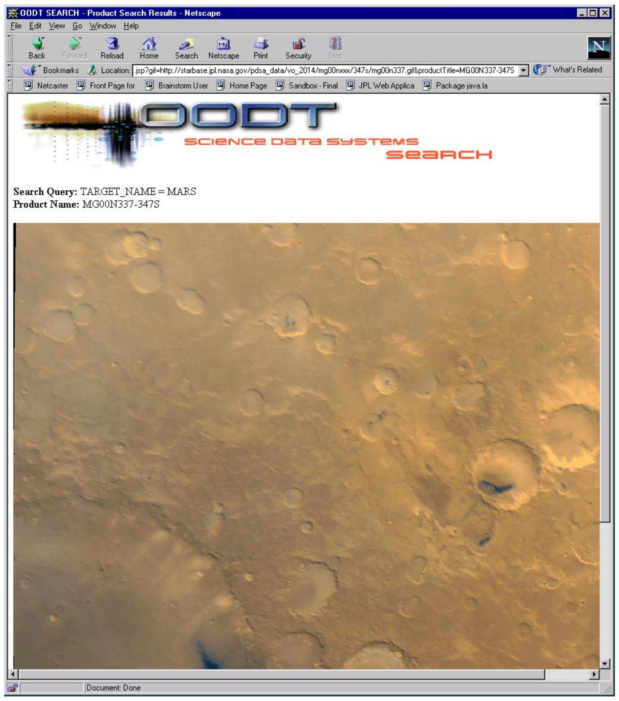
which query handlers must conform. As mentioned, developers creating query handlers define classes that implement the query handler interface allowing the product framework to communicate with the query handlers. Thanks to Java's strong typing and dynamic binding, integrating query handlers is a trivial matter that requires none of the problems typically inherent in the corresponding C/C++ solution of dynamically-loaded libraries. Although not implemented, it is also possible for the product server to alter its set of query handlers during its lifetime without shutting down and restarting the system.

The query handlers are loaded by the product server and passed the XML query. The query handlers transform the queries into the system-dependent query language in order to access the proprietary interface. This moves the responsibility for integrating the data-system–dependent data model onto the data system and away from the OODT data infrastructure. This is an important design consideration for accommodating scalability in a larger enterprise. An example would be the JPL central PDS node. Implementation of a query handler for this node requires that a mapping between the resource location service XML-based query and the central node's Sybase RDBMS be implemented. The query handler would then translate XML-based queries into a SQL-based query referencing the schema that was implemented by the PDS central node. This then provides the core mapping necessary to allow unique data system products to be retrieved from their native environments.

Java's rich standard API, as well as the freely and commercially-available libraries for Java, make developing query handlers a simple matter. For example, accessing data in an SQL database is possible with the JDBC API, part of the standard Java Development Kit. Network resources are available with the networking API. Furthermore, translating the query from XML and product results to XML does not require knowledge of XML since the OODT framework includes the previously mentioned class that encapsulates and manages the XML query document.

The product server framework that is provided is a generic Java-based server that dynamically loads query handlers defined and registered with the service. Once the query handler receives products they must be transformed into a standard format that can be exchanged. The XML query structure defines the result format that allows for data to be returned in various formats. One of the requirements of this architecture is to provide a list of common interchange formats that imposes a set of standards for interoperability. The challenge is to provide a simple set of common formats for images and text, and require that results that fit into these categories use these formats for interchange. This would mean that all images that are in GIF may need to be converted into JPEG if that was the chosen format for images. It is important to point out that results which do not fit into a these standards can be returned in their native format. The goal is to provide flexibility in the architecture, but where possible promote standards for interoperability.

Figure 6 demonstrates an image of Mars returned as a result of a query to a product server using a simple web tool that plugs into the OODT framework.


OODT SEARCH - Product Search Results - Netscape
File Edit View Go Window Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: .jsp?gif=http://starbase.jpl.nasa.gov/pdsa_data/vo_2014/mg00nxxx/347s/mg00n337.gif&productTitle=MG00N337-347S
What's Related
Netcaster Front Page for Brainstorm User Home Page Sandbox - Final JPL Web Applica Package java.la
OODT
SCIENCE DATA SYSTEMS
SEARCH
Document: Done

Search Query: TARGET_NAME = MARS
Product Name: MG00N337-347S

Figure 6: Image Product Query

Products returned in the XML query document take the form of either UTF-8 text (or ASCII text as a proper subset of UTF-8) or as base-64 encoded text, depending on the data type of the product. Although we can return arbitrarily large binary objects with this interface, practical limitations and poorer performance of large

XML document handling suggests another supported alternative: returning a *reference* to the object. For example, a small JPEG JFIF image could be base-64 encoded in the XML query document, whereas a large image could return a URL to access it instead of the image itself in the XML query document. Moreover, the small image could be a thumbnail of the large image, leaving the choice of transferring a large binary object up to the user.

The product server design promotes interoperability by providing an interchange capability to allow a common query mechanism to retrieve products from unique data system implementations. The design presented allows distributed data system nodes to maintain their independence by providing a standard product server that can be extended to access the distributed data systems. This design provides a scalable solution by identifying a standard language for interoperability, and a framework for extending that interoperability to each data system. It also scales by pushing the implementation requirements onto each individual data system.

VI. Conclusion and Future Work

Product servers are presented as components of a distributed framework that allows heterogeneous data systems to communicate and share data. The components of this framework use XML, a standard metadata interchange language that has gained in popularity for improving the ability for applications to be integrated through electronic data interchange. This solution allows for loosely related data systems to remain distributed, while providing a content management and interchange capability for locating specific data products and resources archived at remote locations.

This component framework promotes the use of open standards. This architecture will accommodate changes as XML and standards for interoperability evolve. Currently, many organizations are looking at standards for electronic data interchange and queries using XML. At the time of writing this paper W3C has just published a draft set of requirements for an XML query language [22].

A key to the solution presented is providing data abstraction through the introduction of a distributed framework. Clients are abstracted away from having knowledge of the location and format of individual data products. This is significant since it allows for interoperability across distributed data management systems. In February 1990 the National Science Foundation held a workshop of experts in database management technologies that identified visions for the industry. One vision presented is identified in a report from that workshop that stated "in a multidatabase environment we strive to provide location transparency. That is, all data should appear to the user as if they are located at his or her particular site" [19]. Significant advances in distributed database technology have been accomplished, however, they have focused on distributing components of an integrated homogenous data model in a decentralized architecture. The challenge that the Object Oriented Data Technology task has been investigating is how to introduce an analytical architecture that provides integration of heterogeneous data models and data systems while maintaining location transparency so that the users are unaware that the data is not local.

Metadata provides a foundation to our solution. The ability to map heterogeneous data system implementations is dependent on the ability to understand how to classify and relate the inherent data models behind these systems. Accomplishing these mappings provides the necessary interoperability key needed to relate data products. Advances in Internet and distributed technologies allow these systems to appear as if the data products are local as long as they are network accessible. The solution presented, although applied to planetary, astrophysics, and space science data problems, is not limited to those disciplines. Providing a common metadata model to tie systems together allows for disciplines to begin to build enterprise architectures where data can be shared.

VII. References

1. Aho, A. V., Hopcroft, J. E., Ullman, J. D. Data Structures and Algorithms. Addison-Wesley 1983.

2. Booch et al. The Unified Modeling Language User Guide. Addison-Wesley. 1999.

3. Cook, M.A. Building Enterprise Information Architectures: Reengineering Information Systems. Prentice Hall, 1996.

4. Crichton, D.J., Hughes J.S., Hyon J.J., Kelly, S.C., Science Search and Retrieval using XML, The Second National Conference on Scientific and Technical Data, U.S. National Committee for CODATA, National Research Council, March 13-14, 2000, http://oodt.jpl.nasa.gov/doc/papers/codata/paper.pdf.

5. Crichton, D.J., Hughes J.S., Hyon J.J., Kelly, S.C. Object Oriented Data Technology 1999 Annual Report. Interactive Analysis Environments Program. September 1999. http://oodt.jpl.nasa.gov/doc/reports/annual/1999

6. Data Entity Dictionary Specification Language (DEDSL) - Abstract Syntax, CCSDS 647.0-R-2.0, Draft Recommendation for Space Data System Standards, Consultative Committee on Space Data Systems, November 1999.

7. Deutsch, A., Fernadez, M., Florescu, D., Levy, A., Suciu, D. XML-QL: A Query Language for XML. Submitted to W3C August 19, 1998. http://www.w3.org/TR/NOTE-xml-ql

8. Devlin, B. Data Warehouse from Architecture to Implementation. Addison-Wesley. 1997.

9. Dublin Core Metadata Initiative, Dublin Core Metadata Element Set, Version 1.1: Reference Description, http://purl.oclc.org/dc/documents/rec-dces-19990702.htm.

10. Elmasri,R., Navathe,S., Fundamentals of Database Systems. The Benjamin/Cumings Publishing Company, Inc. 1994.

11. Hughes,J.S., Crichton,D.J., Hyon,J.J., Kelly,S.C., A Multi-Discipline Metadata Registry for Science Interoperability, Open Forum on Metadata Registries, ISO/IEC JTC1/SC32, Data Management and Interchange, January 2000, http://www.sdct.itl.nist.gov/~ftp/l8/sc32wg2/2000/events/openforum/index.htm

12. Hovy, E. Using Ontologies to Enable Access to Multiple Heterogeneous Databases, Open Forum on Metadata Registries, ISO/IEC JTC1/SC32, Data Management and Interchange, January 2000, http://www.sdct.itl.nist.gov/~ftp/l8/sc32wg2/2000/events/openforum/index.htm

13. Inmon, W. H., Zachman, John A., Geiger, Johnthan G. Data Stores, Data Warehousing, and the Zachman Framework: Managing the Enterprise Knowledge. McGraw-Hill. 1997.

14. Marco, David. "Building and Managing the Meta Data Repository". John Wiley & Sons, Inc. 2000.

15. Maruyama, H., Tamura, K., Uramoto, N. XML and Java: Developing Web Applications. Addison-Wesley. 1999.

16. OASIS, Organization for the Advancement of Structured Information Standards, http://www.oasis-open.org.

17. Object Management Group. CORBA/IIOP 2.3.1 Specification. October 1999.

18. Orbacus for C++ and Java version 3.1.3. Object Oriented Concepts, Inc. 1999. http://ooc.com/

19. Stonebraker, M., Hellerstein, J. Readings in Database Systems: Third Edition. Morgan Kaufmann Publishers. San Francisco. 1998.

20. W3C. Document Object Model (DOM), Level 2 Specification. http://www.w3.org/TR/1999/CR-DOM-Level-2-19991210.

21. W3C. Extensible Markup Language (XML), Version 1.0, http://www.w3.org/TR/REC-xml.

22. W3C. XML Query Requirements. W3C Working Draft. 31 January 2000. http://www.w3.org/TR/2000/WD-xmlquery-req-20000131

VIII. About the Authors

Daniel Crichton is a Project Element Manager at JPL, and the principal investigator for the Object Oriented Data Technology task where he is leading a research task developing distributed frameworks for integrating science data

management and archiving systems. He also currently serves as the implementation manager and architect of a JPL initiative to build an enterprise data architecture. His interests are in distributed architectures, enterprise and Internet technologies, and database systems. He holds a B.S. and M.S. in Computer Science. He can be reached at dan.crichton@jpl.nasa.gov.

Steven Hughes is a System Engineer at JPL, and a Co-Investigator for the Object Oriented Data Technology task. He is currently the technical lead engineer for the Planetary Data System and was instrumental in the development of the planetary science meta-model. His interests are in distributed architectures and the role of metadata in interoperability. He holds a B.S. and M.S. in Computer Science. He can be reached at steve.hughes@jpl.nasa.gov.

Sean Kelly is an independent consultant to JPL. He is currently supporting implementation for the Object Oriented Data Technology task and the Enterprise Data Architecture task. His interests include practical applications of software methods and leveraging web technologies in unique ways. He holds a B.S. in Computer Science and a B.S. in Technical Communication. He can be reached at kelly@ad1440.net.

Sean Hardman is a System Design Engineer at JPL. He is currently the System Engineer for the Enterprise Data Architecture task and he supports development of the Object Oriented Data Technology task. His interests include software architecture and database systems. He holds a B.S. in Computer Information Systems. He can be reached at sean.hardman@jpl.nasa.gov.

IX. Copyright

The work described was performed at the Jet Propulsion Laboratory, California Institute of Technology under contract with the National Aeronautics and Space Administration.